Journal of Computer Engineering
ISSN 10003428

Original Research Article

Vol 11, Issue 5, 2022 | ISSN: 1000-3428

Enhancing Scatter Search Metaheuristics through Spark Optimization

Alejandro R. Ferreras, Elena M. Gómez-López

Alejandro R. Ferreras and Elena M. Gómez-López, Departamento de Informática, Universidad de Cantabria, Santander, Spain; Instituto Carnícola de Ciencias Marinas del Atlántico, IIM-CSIC, Spain

Abstract

Optimization problems arise nowadays in all disciplines, not only in the scientific area but also in the field of engineering or economics, and in many others. Currently, challenging optimization problems require solution methods that consume a significant amount of computational resources. The application of High-Performance Computing techniques is a common approach to obtain efficient implementations in traditional parallel computing systems. However, more recent approaches are exploring distributed programming frameworks developed in recent years to achieve efficient computations on clusters and cloud systems. In this paper we present a parallel implementation of the enhanced Scatter Search metaheuristic using Spark. The parallel program was obtained as a particularization of a general software framework we developed to support different realisations of the Scatter Search metaheuristic. The aim of this paper is to provide helpful guidance to readers interested in applying, or developing their own, parallel metaheuristics to solve challenging problems in the Cloud. With the twofold objective of demonstrating the potential of the parallelization with Spark and also of studying the factors that influence the performance of the solution, the proposal has been thoroughly evaluated on two different platforms, a cluster and a cloud platform, using a representative set of parameter estimation problems in the field of Computational Systems Biology.

Indexed keywords: Computer Engineering, Advanced Computing, Technology, Open Access

Article History: Received: 09 February 2022 | Accepted: 10 May 2022 | Published: 20 May 2022

Parameter estimation, Metaheuristics, Scatter Search, Cloud Computing, Spark

1. Introduction

Optimization algorithms are increasingly popular, appearing in many different disciplines and being applied to increasingly large and complex problems. Nowadays, many problems in different areas become extremely challenging and require efficient and robust solution methods based on global optimization [11, 24, 69]. This is the case of parameter estimation problems in Computational Systems Biology, that are used in this work as a case-study to assess our proposal. Computational Systems Biology is the discipline that studies biological systems from an engineering perspective.

© 2022 The Authors. Open Access under CC BY 4.0.

How to cite: Alejandro R. Ferreras, Elena M. Gómez-López (2022). Enhancing Scatter Search Metaheuristics through Spark Optimization. Journal of Computer Engineering, 11(5), 44–69. DOI: https://doi.org/10.5281/zenodo.19344791

In this field, mathematical models are used as a means to help understanding complex biological systems. The

Preprint submitted to Elsevier *June 4, 2020*

construction of these models is an iterative process that starts by proposing a mathematical structure with a set of non-measurable parameters, which have to be estimated to obtain quantitative predictions. The model must then be validated with new experiments and the response can be used recursively in a process of model refinement. Parameter estimation is a key step in this iterative process and can be formulated as a continuous optimization problem subject to a series of dynamic constraints that describe the behaviour of the system over time [59, 8].

Scatter Search (SS) has been shown to outperform many other state-of-the-art metaheuristics for some of these parameter estimation problems [63]. Additionally, an enhanced version of the Scatter Search method (eSS) has been proposed in [18, 19] that overcomes further difficulties arising when dealing with nonlinear dynamic systems. However SS, like many other metaheuristics, is usually time-consuming for realistic problems. The use of High Performance Computing (HPC) techniques may represent an effective strategy to speed up its time to solution. Different parallel implementations can be found in the literature [72, 55, 56] showing good performance for the calibration of several large-scale models.

Taking into consideration the challenging and the dynamic essence of these problems, Cloud Computing represents an interesting approach to the provision and management of the necessary computing resources. *Cloud Computing* [5,
13] is a paradigm for the provision of dynamically scalable infrastructure for application execution and data storage. Clouds are commonly built using virtualisation technologies on resources such as servers, networks or storage, shared by a huge base of users, which contributes to cheapen the computation and storage costs.

Many applications may benefit from the rapidly provision of virtual clusters in the Cloud. This is the case of many key problems in Life Science disciplines, and particularly in Systems Biology, Computational Biology and Bioinformatics [7, 9, 59], with important applications such as, in the field of medicine, knowing how pathogenic cells interact to develop new therapies against, for instance, cancer or metabolic or autoimmune diseases. However, until now most research efforts involving Cloud Computing in these fields have been focused on capturing, storing and comparing large volumes of data, rather than on exploiting the Cloud potential to solve compute-intensive Life Science brainteasers [60]. As traditional HPC approaches do not reach their full potential when used in the Cloud, new strategies and frameworks have still to be explored in this domain.

In this work we explore the use of Spark [77] to implement a cloud-oriented parallel version of the eSS metaheuristic. Spark is a throughput-oriented framework that outperforms other cloud solutions by adding improved support for iterative algorithms through in-memory computing. Spark provides essential features to address the challenges of large-scale distributed applications: high-level programming models to facilitate user program parallelization, support for data distribution and multi-node/core processing, and runtime features such as fault tolerance and load balancing. We have evaluated the performance of the proposal using a set of demanding parameter estimation problems in the field of Computational Systems Biology. Note, however, that the applicability of the methodology proposed is broader, and could be applied to many other challenging fields.

The rest of the paper is structured as follows. Section 2 summarises related work. Section 3 briefly describes the SS and eSS metaheuristics used as a basis for this work. The parallel implementation of the eSS proposed in this paper is described in detail in Section 4, while Section 5 presents exhaustive experimental results to demonstrate its efficiency in parameter estimation problems in Computational Biology systems. Finally, Section 6 concludes the paper.

2. Related Work

2.1. *Performance of HPC applications in the cloud*

Though Cloud Computing technologies represent a powerful approach to managing technical computing resources, some challenges still remain for the adoption of cloud in HPC applications [60]. The most important are security and performance. Security is still an important barrier to acceptance, however, the problem lies mainly in the trust and perception of users, rather than limitations in capability and architecture of various cloud platforms. Regarding performance, several researchers have studied the performance of HPC applications in cloud environments. Traditional HPC benchmarks have been used in those analysis, such as the NAS benchmarks in [22], the Linpack benchmark in [50], or a range of microbenchmarks and HPC kernels in [53]. Real applications have also been evaluated in the cloud, such as bioinformatic applications [32], high-energy and nuclear physics experiments [37], and different e-Science applications [58, 41]. A set of representative applications running at a supercomputing center have been employed in [34]. In addition, a comprehensive analysis to identify, for HPC applications in the cloud, the more hazardous problems and bottlenecks can be found in [23]. All these works conclude that the lack of high-bandwidth, low-latency networks, as well as the virtualization overhead, have a significant impact on the performance of HPC applications in the cloud. Reacting against these issues, some cloud providers, such as Amazon [4] or Microsoft Azure [49], have recently provided compute nodes that use hardware found in HPC clusters and claim to be optimized for running HPC applications.

Among the new programming models intended to deal with large scale computations on cloud systems, MapReduce [16] is the most popular one. MapReduce executes in parallel several instances of a pair of *map* and *reduce* functions, provided by the user, on a distributed set of *worker* processes driven by one *master* process. In MapReduce executions are run in batches, using its distributed filesystem, called HDFS, to take the input and store the output. A wide range of applications have been parallelized using MapReduce, but, when applied to iterative algorithms this programming model exhibits significant performance bottlenecks [21] typically because there is no efficient way of reusing data or computation from previous iterations. New proposals, like Spark [76], aim to yield an efficient solution for iterative algorithms. Compared to MapReduce, the performance of iterative algorithms using Spark can be improved by an order of magnitude, according to [76].

Furthermore, projects like DataMPI [46] or CloudMPI [1] appeared in an effort to connect cloud platforms and HPC. DataMPI [46] tried to extend MPI by key-value pair based communication operations for cloud platforms. The cloudMPI [1] framework attempted to design and implement an MPI-like solution for cloud infrastructures, being the Azure cloud platform

their preliminary testbed. Unfortunately, these projects achieved a modest success, and they are not active anymore.

2.2. *Parallel implementations of the Scatter Search metaheuristic*

In an attempt to reduce the time for solving very challenging optimization problems, the parallelization of metaheuristics has been large studied in the last decade [3]. Many parallel algorithms have been proposed, some of them focused on the parallelization of the Scatter Search. In [26] three parallel strategies were explored: a low-level synchronous parallel SS model using parallel search instead of local search, a replicated combination SS model that distributes multiple subsets on the processors, and a natural replication of parallel SS. All these methods were implemented using shared-memory techniques, and, thus, present limitations on scalability. In [12] a parallel algorithm based on SS and path re-linking methods was presented. In this proposal, the master process creates the starting solutions set while calculations of path re-linking are executed by the slave processes on local data. The slaves send the new solutions to the master that creates a new set of starting solutions. Another parallel SS algorithm is presented in [45], based on replacing the combination method by parallel execution of two greedy methods on every processor. In [72], a cooperative parallel strategy for the eSS method is proposed. This method supports an island-based approach implemented in a master-slave means, where a sequential eSS algorithm (island) is run in each slave, exchanging their set of solutions through the master at certain fixed time moments. The cooperation of the individual islands modifies the systemic properties of the algorithm, improving its performance and outperforming both the sequential Scatter Search and other encouraging metaheuristics. However, the synchronization among slaves causes a poor scalability when the number of processors grows. In [55], an asynchronous cooperative parallel strategy (aCeSS) is proposed to improve the CeSS algorithm by means of a cooperative scheme driven by quality of solution, instead of elapsed time. However, the communication protocol and the double-ring topology proposed still compromise the efficiency of the solution. In [56] a self-adaptive cooperative enhanced Scatter Search, implemented a hybrid MPI+OpenMP strategy, is proposed demonstrating encouraging results for solving very large and hard optimization problems.

All of these proposals are parallel implementations based on traditional parallel programming interfaces. To the best of our knowledge, there is no report of any cloud-oriented parallel implementation of Scatter Search, nor any assessment of traditional parallel implementations in cloud infrastructures.

2.3. *Parallel metaheuristics in the cloud*

Cloud-oriented parallel metaheuristics have also received increasing attention in the last decade. Most of the works in the literature, though, are based on MapReduce. The parallelization of Genetic Algorithms (GAs) can be found in [71], that attempts to fit the GAs into the MapReduce model. According to [35] GAs cannot be easily parallelized using MapReduce due to their specific features, so they propose to incorporate a hierarchical reduction phase to overcome this issue. Nevertheless, since they only perform in parallel the fitness evaluation the results obtained showed a poor scalability. Another attempt to scale the population of GAs by using MapReduce can be found in [33]. The Particle Swarm Optimization (PSO) has been parallelized in [48] using

MapReduce. In [78] the fitness evaluations in the Differential Evolution (DE) algorithm are executed in parallel using Apache Hadoop (the most popular MapReduce framework). The experimental results show that the additional cost of Hadoop DFS I/O operations and the system overload significantly narrow the benefits of parallelization. In [57] the use of MapReduce to parallelize the Simulated Annealing (SA) algorithm was also explored. Different algorithmic patterns of distributed SA with MapReduce were designed and evaluated on the Azure public cloud. In [75] Hadoop is used to scale the parallelization of an Ant Colony Optimization (ACO). In [40] a MapReduce hybrid GA-PSO optimization framework to infer large gene networks is proposed. Recently, a novel algorithm using Hadoop to solve clustering problems by means of a parallel bat algorithm (PBA) is proposed in [6].
The use of Spark for the parallelization of different metaheurisitics is also a research trend in recent years. In [15]

Spark has been tested to solve the Job Shop Scheduling Problem (JSP) using the coral reff optimization algorithm (CRO). The use of Spark to parallelize the DE algorithm was explored in [65]. And a comparison of this Spark implementation with a MapReduce implementation has been shown in [67], concluding that Spark outperforms MapReduce in this kind of iterative algorithms. In [47] a proposal that improves the performance of the *k*-means algorithm through the use of a tabu search and the parallelization of clustering through Spark is presented. Finally, a recent study [68] presented a simple framework to show how to use Spark to scale metaheuristic algorithms for clustering problems.

Though there have been large efforts assessing the performance of specific programming models or frameworks in different computing platforms using standard benchmarks, there are few studies on evaluating and discussing the performance of a particular kind of application using different models and platforms. The performance and data management of Spark and MPI/OpenMP on Google Cloud platform were compared in [62]. Experiments with a particle physics data set show that MPI/OpenMP outperforms Spark by more than one order of magnitude. Even so, it is important to notice also that Spark offers a better data management infrastructure and it incorporates other important aspects such as the posibility of dealing with node failures and data replication. In [27] Spark is compared to traditional MPI implementations in matrix factorization problems. The conclusions of this work show that there is still room for improving the Spark performance. However, this work only includes experiments in supercomputers. In [66] the implications of the use of MPI and Spark in the parallelization of the DE algorithm were also explored. The exhaustive experimental discussion includes experiments in different computing platforms, including public clouds, and the differences that appear from the innate features of each programming model were discussed. Recently, three machine learning algorithms implemented in MPI, Spark, and Flink were assessed in [36] that compares their performance and identifies strengths and weaknesses in each platform.

3. Scatter Search Algorithm

The Scatter Search (SS) metaheuristic [30] is a population-based algorithm that applies strategies for search diversification and intensification that have proved effective in a variety of optimization problems. It originates in integer

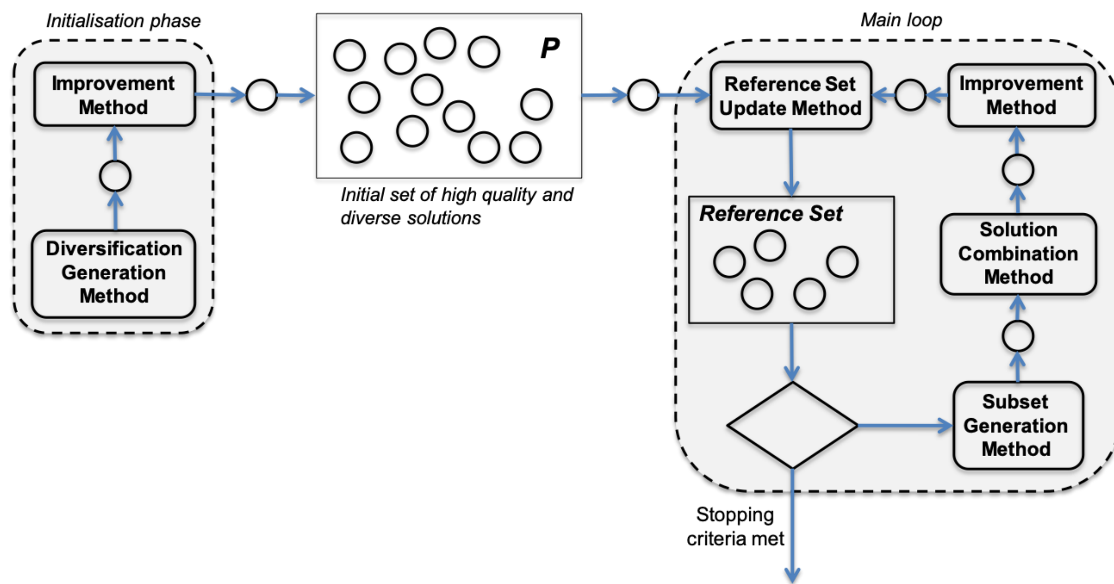


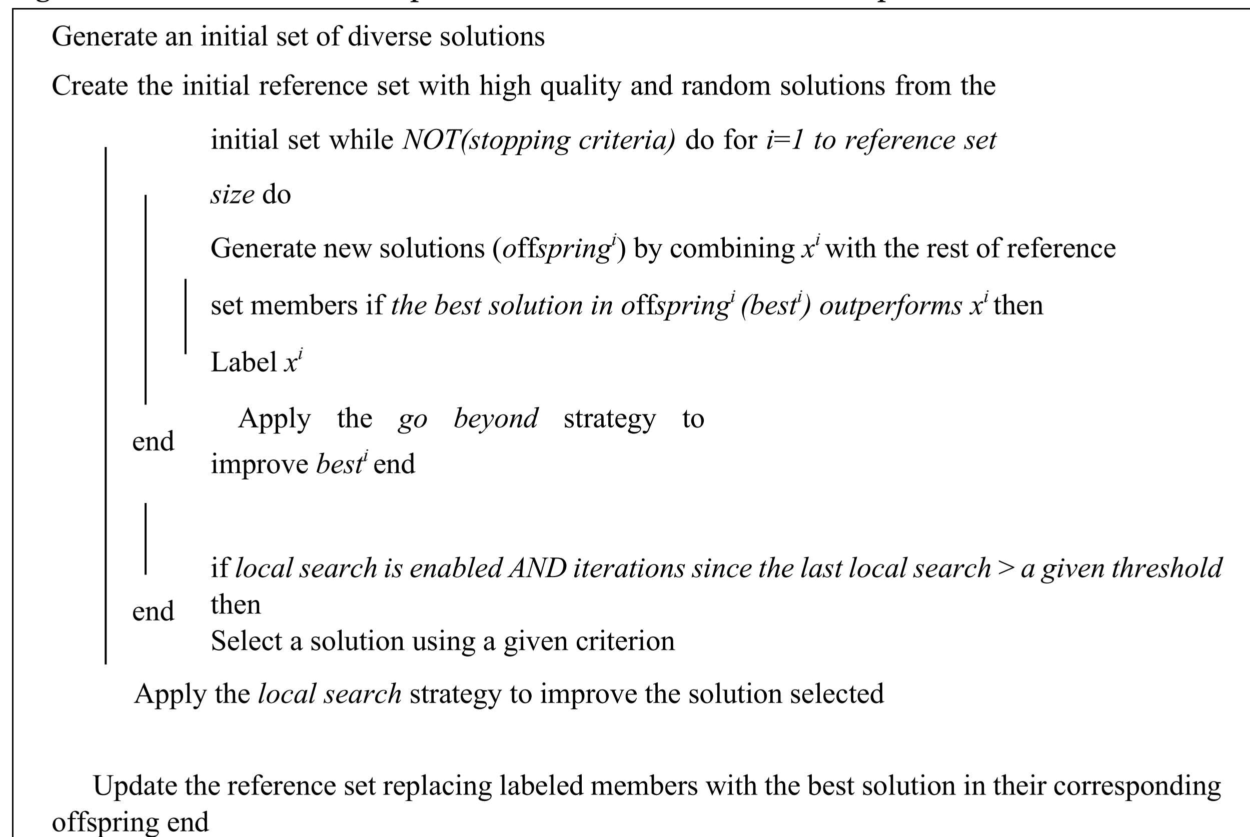
Figure 1: The five-method Scatter Search template.

programming formulations for combining decision rules and problem constraints [28]. Like many other evolutionary algorithms, the goal of the method is to improve the quality of the solutions by subsequent combinations and replacements. Compared to other evolutionary methods, SS uses a smaller set of quality population members, called the *reference set* and incorporates an *improvement method*, usually a *local search*, to expedite the convergence of the algorithm.

The SS metaheuristic is very flexible, since each of its components can be implemented in different ways to adapt the method to a variety of problems. The well-known *five-method template* proposed in [29] has served as the main reference for most of the SS implementations to date:

1. A *Diversification Generation Method* (DG) to generate a collection of diverse trial solutions.
2. An *Improvement Method* (IM) to transform a trial solution into one or more enhanced trial solutions.
3. A *Reference Set Update Method* (RSU) to build and maintain a reference set consisting of a small set of the best solutions found. Solutions gain membership to the reference set according to their quality or their diversity.
4. A *Subset Generation Method* (SG) to operate on the reference set, to produce several subsets of its solutions as a basis for creating combined solutions.
5. A *Solution Combination Method* (SC) to transform a given subset of solutions produced by the SG method into one or more combined solutions.

Figure 1 shows an schematic representation of the five-method template.



Generate an initial set of diverse solutions

Create the initial reference set with high quality and random solutions from the

initial set while *NOT(stopping criteria)* do for *i=1 to reference set*

size do

Generate new solutions (*offspring*^{i}) by combining x^i with the rest of reference

set members if *the best solution in offspring*^{i} *(best*^{i}*) outperforms* x^i then

Label x^i

end Apply the *go beyond* strategy to

improve *best*^{i} end

if *local search is enabled AND iterations since the last local search > a given threshold*

end then

Select a solution using a given criterion

Apply the *local search* strategy to improve the solution selected

Update the reference set replacing labeled members with the best solution in their corresponding offspring end

Algorithm 1: Pseudocode of the enhanced Scatter Search algorithm

In this paper we will focus on the enhanced Scatter Search (eSS) method proposed in [18, 19]. The eSS presents a more effective pattern that aims to get over frequent issues of nonlinear dynamic systems optimization such as noise, flat areas, non-smoothness, and/or discontinuities. Algorithm 1 shows the pseudocode of the eSS method. This method implements original procedures in different regions to balance intensification, that is, local search, and diversification, that is, global search. It uses a small population size without memory structures (repeated sampling is allowed) even for large-scale problems but allows more search directions, compared to the classical SS, due to a novel combination strategy based on wide hyper-rectangles. The diversity in the search is maintained, and the number of evaluations per iteration does not get larger. It also incorporates a search intensification procedure (the *go-beyond* strategy) to exploit the encouraging directions generated by combination of solutions. The eSS method comes up with a good balance between robustness and efficiency in the global phase, incorporating a local search strategy to expedite the convergence of the algorithm. Nonetheless, eSS still requires unreasonable computation times for most complex problems, like those arising from parameter estimation in dynamic models.

4. Parallel implementation of the enhanced Scatter Search

In this section we present our proposal for a parallel implementation of the eSS metaheuristic presented in [18, 19] using Spark. This section is divided into three parts: in subsection 4.1 the implementation of the SS as a flexible software framework is introduced, in subsection 4.2 the particularisation of that framework to obtain an instance of

Inputs: PG (*population generation method*); PE (*population evolution method*); TC (*termination criteria*) Output: the best solution found *population* = apply the PG method

while *NOT(TC)* do | apply the PE method to *population* end return the best individual in *population*

Algorithm 2: The general algorithm of a population-based metaheuristic.

the eSS is detailed, and in subsection 4.3 the Spark-based parallel implementation of the eSS instance is explained.

4.1. *The Scatter Search framework*

As we have seen in Section 3, SS has been described from an algorithmic point of view as a general template that can be particularised for a wide class of problems by means of specific realisations of a reduced set of methods [29, 61]. Each implementation of the SS is an instance of that general template in which different intensification and diversification strategies are combined to obtain a version of the metaheuristic conveniently adapted to deal with a specific class of optimization problems. To support that level of abstraction we have implemented a flexible software framework that can be easily adapted to obtain different realisations of the SS.

As our primary interest is to experiment with new cloud programming models for the parallel implementation of metaheuristics, we have adopted Scala [51] as the programming language of our framework because it is the programming language in which distributed frameworks like Spark or Flink are coded. Scala is a concise, very expressive high-level language, that integrates common features of object-oriented and functional languages. Using Scala has allowed us to capture the required level of abstraction to implement SS. In the rest of this subsection we describe the main ideas behind the design and implementation of our software framework.

SS shares a common scheme with other population-based metaheuristics like, for instance, evolutionary algorithms, differential evolution or particle swarm optimization. All these methods have in common that they generate an initial population that is repeatedly evolved until the termination criteria are met. We have captured that common scheme as a general algorithm (Algorithm 2) applying the *template method* design pattern implemented as a *higherorder function* (i.e. a function that takes other functions as arguments). To instantiate a particular population-based metaheuristic, three methods has to be provided as inputs to this general algorithm: the method to generate the initial population (PG), the method to evolve a given population in the search of higher-quality individuals (PE) and the termination criteria (TC) to

stop the search and return the best individual found. Each of these methods can be itself a *higher-order function*, so concrete instances of the algorithm can be as complex as needed.

To illustrate how this general algorithm is particularised to get a concrete metaheuristic, we'll use the SS PG method as an example. SS starts by "*generating a starting set of solution vectors by heuristic processes designed for*

```
Inputs: DG (diversification generation method); IM (improvement method); RSU (reference
set update method)
Output: the initial population subset of reference solutions
(reference set) initial = apply the DG method population = apply
the IM method to initial reference set = apply the RSU method to
population return reference set
```

Algorithm 3: The general algorithm of the Scatter Search population generation method.

the problem considered, and designate a subset of the best vectors to be reference solutions" [29]. To accomplish that, three methods are involved (Figure 1): the *diversification generation* (DG) method to generate a collection of diverse trial solutions, the *improvement* (IM) method to transform a trial solution into one or more enhanced trial solutions and the *reference set update* (RSU) method to build the initial reference set. Algorithm 3 shows the general algorithm of the SS PG method, which is again an application of the *template method* design pattern implemented as a *higher-order function*. To obtain a specific instance of the SS PG method, implementations of the DG, IM and RSU methods must be provided as inputs to the general algorithm (note that *higher-order functions* could be used again in case complex implementations were required).

There are many possible implementations of the DG, IM and RSU methods that will lead to different instances of the SS PG method. Our framework support methods with different implementations by applying the *Strategy* design pattern implemented using *function builders* (i.e. higher-order functions that return a function [10]). The selection of a specific method implementation is configured in a properties file that is read at the beginning of the execution. Table 1 shows a summary of the properties, values and their meaning for the implementations of the DG, IM and RSU methods currently supported in our framework's configuration file. Note that an implementation of NL2SOL [17], a local solver that has previously demonstrated its effectiveness for the parameter estimation problems used as benchmarks in the experimental section [20], has been integrated in the framework as one of the options for the IM method. With the values shown in Table 1 a total of eight different instances of the SS PG method are posssible: PG(DG=Random, IM=NL2SOL, RSU=NBest/NRandom), PG(DG=Random, IM=None, RSU=NBest/NDiverse), etc.

The example we have just seen has served to show the basic design principles we have applied to implement a flexible software framework that supports different realisations of the SS. Although a small set of method implementations have been used in the example, it goes without saying that the framework has been designed with extensibility in mind to facilitate the addition of more method implementations. In the next subsection we'll explain how an implementation of the eSS has been instantiated using this framework.

Table 1: Properties of the DG, IM and RSU methods.

method	property	value	description	parallel
DG	solution generator	Random	initial solutions are generated randomly	X
		Fixed	initial solutions are predetermined	
IM	initial improvement	None	initial population is left as it is (no improvement)	
		NL2SOL	initial solutions are improved using the NL2SOL local solver	X
RSU	refset create	NBest/NRandom	one half of the solutions in the reference set are the best solutions of the initial population and the other half are selected randomly from the remaining solutions in the population [18, 19]	
		NBest/NDiverse	one half of the solutions in the reference set are the best solutions of the initial population and the other half are the most diverse of the remaining solutions in the population	

4.2. *Implementation of the enhanced Scatter Search*

Applying the same design principles explained in the previous subsection using the SS PG method as example, the other two methods in Algorithm 2 have also been particularised. Algorithm 4 shows the general algorithm of the SS PE method which corresponds to the body of the SS main loop shown in Figure 1 with a slight modification to include the intensification strategy proposed in the eSS (Algorithm 1). To that end, an optional *local search* (LS) method has been added just before returning the updated reference set.

Table 2 shows a summary of the properties, values and their meaning for the implementations of the input methods (i.e. SG, SC, IM, RSU and LS) of the SS PE general algorithm currently supported in our framework's configuration file. Notice that the implementation of the SG method has been restricted to generate only pairs of solutions because in [39] it has been checked that most of the quality solutions obtained by combination arise from sets of two solutions. Moreover, as the LS method can be used to include complex intensification strategies, it has been represented in our framework as a general algorithm that can be particularised to different LS instances, as we have already seen with other methods[1]. Currently, a basic tabu search that can be particularised with different local solvers and strategies for deciding when and to which solutions to apply the local solver, is the only implementation available for the LS method. NL2SOL was the local solver used for the tabu search in the experiments reported in this paper, since it has been found very effective for this class of problems [73, 25].

[1] A detailed explanation of the LS method is left out of the scope of this paper due to size restrictions.

```
Inputs : RS (the reference set); SG (subset generation method); SC (solution combination
method); IM
        (improvement method); RSU (reference set update method); LS (local search,
        optional)
Output: the updated reference set subsets = apply the SG
method to RS offspring = apply the SC method to subsets
improved = apply the IM method to offspring updated =
apply RSU to update RS with solutions in improved
if LS is defined then |
apply LS to updated
end return updated
```

Algorithm 4: The general algorithm of the Scatter Search population evolution method.

With regard to the *termination criteria* (TC) in Algorithm 2, it has been implemented using an OR composition of boolean functions. Every function in the composition is instantiated from the configuration specified in the same properties file that is used to select the concrete implementations of the methods. Currently, conditions to limit the number of evaluations of the fitness function, the maximum execution time or the fitness value to reach are supported and the framework has been designed to facilitate the addition of new conditions.

Algorithm 3 and Algorithm 4, in combination with the implementations described in Table 1 and Table 2, provide what is needed to obtain a reduced set of SS instances that could be increased by adding more method implementations to the framework. Specifically, the eSS proposed in [18] can be instantiated using the following configuration: {PG(DG=Random, IM=None, RSU=NBest/NRandom), PE(SC=HyperRectangle, IM=GoBeyond, RSU=1by1, LS=TabuSearch), TC(max evaluations, max time, value to reach)}.

4.3. *Parallelisation of the enhanced Scatter Search*

So far we have seen how the eSS has been implemented as an instance of a general framework that provides support to different SS realisations. In this subsection we'll explain the parallel implementation of an eSS instance using Spark.

Our approach has started by running some preliminary tests to identify the most costly methods, in terms of execution time, of the eSS. These methods are the ones with a tick[2] in the column *parallel* of tables 1 and 2. Not surprisingly, they correspond to methods that include the fitness evaluation of new solutions.

Then, we have developed Spark-based parallel implementations of those methods and we have added them to our framework. All the implementations follow a *master-slave* model of parallelisation in which the data that will be distributed from the master to the slaves are represented using *resilient distributed datasets* (RDDs, [76]). The

Table 2: Properties of the SG, SC, IM, RSU and LS methods (n/a = not applicable).

[2] In this paper the parallelisation of the LS method has not been considered.

method	property	value	description	parallel
SG	n/a	n/a	generates all pairs of reference solutions (subsets of size 2)	
SC	subset combination	Linear	new solutions are obtained by linear combination	X
		HyperRectangle	new solutions are obtained by applying the *hyperrectangles-based combination method* [18, 19]	X
IM	improvement	GoBeyond	solutions are improved by applying the *go-beyond intensification strategy* [18, 19]	X
RSU	refset update	1by1	the reference set is updated by applying the *(1+1) updating strategy* [19] (i.e. a solution can only enter the population by replacing its parent)	
LS	local_search	TabuSearch	a *tabu search* implementation that combines a local solver, a tabu list and methods for determining when and with which solutions to start the local search	

RDD abstraction is used in Spark to represent read-only fault-tolerant partitioned collections of records that can be manipulated using a rich set of operators like *map*, *filter* or *join*. The Spark *driver* (the master in Spark terminology) partitions RDDs and distributes the partitions to *workers* (the slaves in Spark terminology). Workers persist RDDs (in memory by default), transform them by applying the same operations to many data items at the same time and return the results to the driver when actions like *count*, *reduce* or *collect* are executed.

Figure 2 shows a representation of the Spark-based parallel implementation of the SS initialisation phase (Figure 1). In the figure, the boxes with solid outlines and shaded rectangles in the inside are RDDs (the shaded rectangles represent the RDD partitions). The operations that have been distributed are:

- In the DG method, the fitness evaluation of the initial set of diverse solutions, implemented as a Spark map

  transformation.

- In the IM method, the improvement of the initial set of diverse solutions, also implemented as a Spark map

  transformation.

These operations are usually very time-consuming for the type of challenging problems we are dealing with, because they involve the evaluation of complex fitness functions and the application

of local solvers. Notice that this parallel implementation is valid for any instance of the SS. For the specific realisation of the eSS proposed in [18] the second map would not be applied.

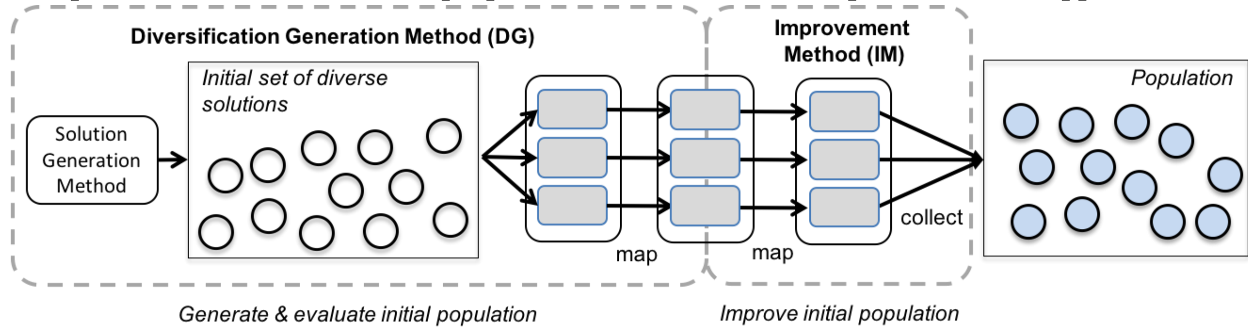


Figure 2: Spark-based parallel implementation of the Scatter Search initialisation phase.

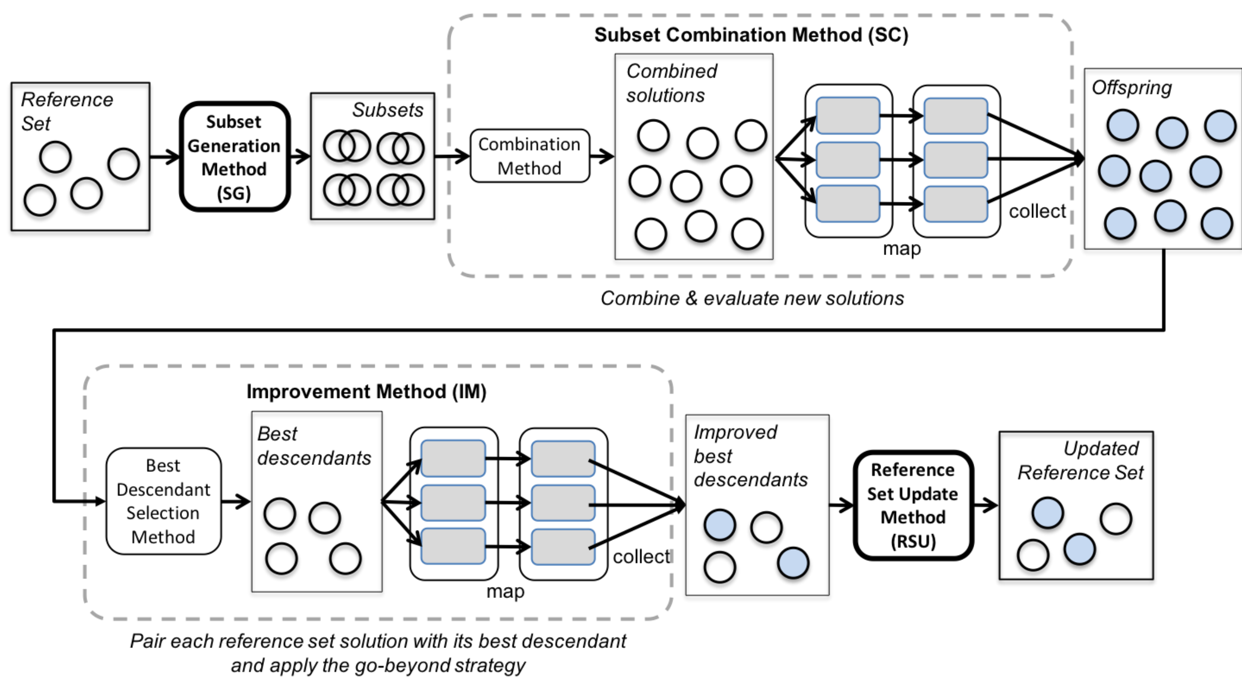


Figure 3: Spark-based parallel implementation of the enhanced Scatter Search main loop (with LS not defined).

Figure 3 shows the representation of the Spark-based parallel implementation of the body of the eSS main loop

(Figure 1). In this case, the operations that have been distributed are:

- In the SC method, the fitness evaluation of the new solutions obtained by applying the combination method (the *hyper-rectangle-based combination method* in the eSS), implemented as a Spark map transformation.

- In the IM method, the improvement of the best descendants, also implemented as a Spark map transformation. In the eSS, the *go-beyond strategy* is applied whenever a reference set solution is outperformed by the best of its descendants. This has been implemented by using an RDD of *reference set solution-best descendant* pairs that is distributed to the workers to apply the *go-beyond strategy* to each pair in parallel.

To instantiate the Spark-based parallel version of the eSS, the same configuration of the sequential instance (subsection 4.2) and an additional *execution environment* object have to be defined in the framework's properties file.
Currently, the framework supports sequential (used by default) and Spark execution environments.

5. Experimental results

In order to evaluate the efficiency and scalability of the Spark-based parallel implementation of the eSS, the problem of parameter estimation in the domain of Computational Systems Biology was considered as an example case. Note, however, that the method is general and can be applied to optimization problems in other domains.

Parameter estimation problems in dynamic models are described by deterministic nonlinear ordinary differential equation models. Given a model an a measurements data set, the goal of the estimation is to find the optimal unknown model parameters that minimizes the variance between model predictions and measurements. This variance is given by a cost function that quantifies the model error. Assuming a generalized least squares cost function, the mathematical statement corresponds to the nonlinear programming (NLP) problem of finding vector p that minimizes:

$$J = \sum_{\varepsilon=1}^{n_\varepsilon}\sum_{o=1}^{n_{\varepsilon_o}}\sum_{s=1}^{n_{\varepsilon,s}^{o}} (ym_{\varepsilon,s}^{o} - y_{\varepsilon,s}^{o}(p))^T W (ym_{\varepsilon,s}^{o} - y_{\varepsilon,s}^{o}(p)) \quad (1)$$

where n_ε is the number of experiments, n_ε is the number of the state variables measured experimentally, $ym^{\varepsilon,}{}_{s}{}^{o}$ corresponds with the measured data, $n_{,s}{}^{o}$ is the number of the samples per state variable for each experiment, $y^{\varepsilon,}{}_{s}{}^{o}(p)$ are the model predictions, and W is a scaling matrix that balances the residuals.
Additionally, the optimization of these problems is subject to some constraints:

$$\dot{x} = f(x,p,t) \quad (2) \qquad x(t_o) = x_o \quad (3) \qquad y = g(x,p,t) \quad (4)$$

$$h_{eq}(x,y,p) = 0 \quad (5) \qquad h_{in}(x,y,p) \leq 0 \quad (6)$$

$$p^L \leq p \leq p^U \quad (7)$$

where f is the set of differential-algebraic equations (DAEs) that define the dynamics of the process, x is the vector of state variables and x_o are their initial conditions, g corresponds with the observation function, h_{eq} and h_{in} are equality and inequality constraints, and p^L and p^U are upper and lower bounds for p.

The experimental results reported in this work correspond to the following parameter estimation problems:

Table 3: Parameters and some additional features of the problems used in the experiments.

benchmark	# initial solutions	reference set size	# max. evals[a]	eval. time (s)[b]	# eval./iter. [c]	# iter.[d]
Circadian			1 000 000	0.00454545		7575
Nfkb			300 000	0.02870657		
B2	1160		180 000	0.10274621	1277	
B4	1170		1 000 000	0.00698274	1261	

[a]Stopping criterium: maximum number of evaluations [b]Average time of one fitness evaluation in the local cluster

[c]Maximum number of evaluations in each iteration of the main loop (Figure 3)

[d]Maximum number of iterations of the main loop

- *Circadian* model: parameter estimation in a dynamic model of the circadian clock in the plant *Arabidopsis thaliana*, as presented in [44]. The model consists of 7 ordinary differential equations with 27 parameters (13 of them were estimated) with data sets from 2 experiments.
- *NFKB* model: this problem is based on the model in [43] and consists of 15 ordinary differential equations with 29 parameters and data sets from 2 experiments.
- Two benchmarks from the BioPreDyn-bench suite [74]:
  - *Problem B2*: a dynamic model of the central carbon metabolism of *E. coli*. It consists of 18 dynamic states, 9 observed states and 116 estimable parameters.
  - *Problem B4*: a kinetic metabolic model of Chinese Hamster Ovary (CHO) cells, with 34 dynamic states, 13 observed states and 117 parameters.

These problems were selected because they range from small problems, such as Circadian and NFKB, to largescale problems such as B2 or B4. In both subsets, there is one problem in which the cost function computation is very fast (computationally) and another in which it is slow. There are also problems that perform few evaluations per iteration and others that perform many. In summary, these four problems have been chosen because they compose a representative benchmark for a thorough evaluation of our proposal. Table 3 shows a brief description of the parameters used for the experiments and some additional features that will be used later to explain the results.

Guidance of [31, 2] was followed in the conception of the experiments reported in this section. In order to perform a statistical analysis of the results, each experiment was executed a number of 20 independent runs. Since the main goal of this work is to evaluate the potential of Spark in the parallelization of the SS algorithm, the focus is placed on calculating the acceleration achieved by performing the evaluation of the fitness function in parallel. So, a vertical cut that evaluates the performance of a fix number of fitness evaluations, i.e. a predefined effort, was used to that end.

For the experimental testbed two different platforms were used. On the one hand, experiments were conducted in our local cluster called *Pluton*, that consists of 16 nodes powered by two octa-core Intel Xeon E5-2660@2.60 GHz CPUs with 64 GB of RAM, and connected through an InfiniBand FDR network. On the other hand, the same set of experiments was carried out using the resources of the Galician Supercomputing Center (CESGA) [14], specifically its cloud computing platform, based on OpenNebula v5.4.6 [52] which we will refer to as *Nebula* from now on. We replicated an environment similar to that of our local cluster, deploying several virtual machines with 16 virtual cores each on a cluster of AMD Opteron 6174 based servers.

Median execution time and speed-up results using a maximum effort as stopping criterion are shown in Table 4. This table displays, for each benchmark, the number of cores (*#np*) used, the median best value achieved (*fbest*), the median execution time of all the runs in the experiment, the speed-up (calculated as T_{seq}/T_{par}), and the efficiency (calculated as *speed-up*/*np*). To help us to discuss in detail the results obtained, their relation with the parameters of each benchmark and the performance of each platform, the results in Table 4 are graphically shown in Figure 4. Additionally, we have also calculated the number of evaluations per second and per core (evals/s/core) for each benchmark on both platforms (Figure 5). This is a good metric for evaluating the implementation because it includes not only the CPU time of the evaluations, but also the communications and overhead time.

From these results, several conclusions can be drawn. First, the performance on *Nebula*, in terms of the execution time required to carry out the experiments, is substantially impacted. This can be explained by the virtualization overhead, the differences in the underlying hardware platforms and the overhead due to the use of non-dedicated resources. Second, whether in the local cluster or on the cloud platform, the parallel implementation outperforms the sequential version, achieving promising speed-ups for a small number of cores. Third, the scalability of this parallelization is limited, although it depends strongly on the problem at hand.

The Circadian is the problem that achieves the lowest speedup and, more specifically, the poorest scalability. The reasons are found in the shortest time needed to evaluate the fitness function and in the small number of evaluations that are performed in each iteration of the main loop of this benchmark (Table 3). For each iteration of the main loop (Figure 3), the evaluation and improvement of the new solutions is distributed among the available cores. In the case of Circadian, with a reference set of only 12 individuals, there are few evaluations to distribute in each iteration (e.g. for 16 cores, each core would perform at most 9 evaluations). Moreover, because the time needed for each evaluation is very short, this benchmark performs a high number of iterations which increases the communication overhead. Thus, the efficiency drops quite early, for 8 cores it is already below 50%. Moreover, the impact of these features on the scalability is higher on *Nebula* since the performance of this platform is lower. In addition, this is

also the reason why the number of evaluations per second in this benchmark is dropping the most quickly as the number of cores increases (Figure 5).

At the other end, the B2 benchmark presents the largest fitness evaluation time and the largest number of evaluations per iteration. Its results are better, both in terms of speedup and scalability, because it has more computational load to do in each iteration, thus performing the lowest number of iterations and introducing less communication

Table 4: Results of the experiments: Median best value, median execution time in seconds, speed-up and efficiency.

benchmark	#np	fbest		time (s)		speed-up (efficiency)	
		Pluton	Nebula	Pluton	Nebula	Pluton	Nebula
Circadian		0.58799	0.74189	4979	9210	—	—
		0.63741	0.74805	3078	5299	1.62 (81%)	1.74 (87%)
		0.95049	0.70488	1893	2996	2.63 (66%)	3.07 (77%)
		0.66422	0.58749	1360	2491	3.66 (46%)	3.70 (46%)
		0.94249	0.87076	1216	2404	4.10 (26%)	3.83 (24%)
		0.75451	0.87472	1036	2782	4.81 (15%)	3.31 (10%)
		0.62465	1.08300		3148	5.21 (8%)	2.93 (5%)
		0.83447	0.70016	1063	3229	4.68 (4%)	2.85 (2%)
Nfkb		0.045	0.047	8376	18034	—	—
		0.046	0.046	5012	9599	1.67 (84%)	1.88 (94%)
		0.044	0.047	2568	5401	3.26 (82%)	3.34 (83%)
		0.046	0.044	1469	3199	5.70 (71%)	5.64 (70%)
		0.046	0.044		1929	9.52 (60%)	9.35 (58%)
		0.046	0.046		1244	14.45 (45%)	14.50 (45%)
		0.045	0.046			21.90 (34%)	19.97 (31%)

	0.046	0.046			27.24 (21%)	22.06 (17%)
B2			19024	33738	—	—
			10594	19211	1.80 (90%)	1.76 (88%)
			5872	10117	3.24 (81%)	3.33 (83%)
			3428	5803	5.55 (69%)	5.81 (73%)
			2136	3242	8.91 (56%)	10.41 (65%)
			1327	1985	14.34 (45%)	17.00 (53%)
				1272	21.40 (33%)	26.53 (41%)
					30.08 (23%)	38.80 (30%)
B4	25541	18402	7213	12931	—	—
	21806	21230	4340	7301	1.66 (83%)	1.77 (89%)
	23055	26554	2285	4243	3.16 (79%)	3.05 (76%)
	26726	20742	1274	2436	5.66 (71%)	5.31 (66%)
	18823	26081		1671	8.13 (51%)	7.74 (48%)
	21234	24555		1090	12.80 (40%)	11.86 (37%)
	25519	21637			17.74 (28%)	15.16 (24%)
	25961	21087			20.38 (16%)	15.93 (12%)

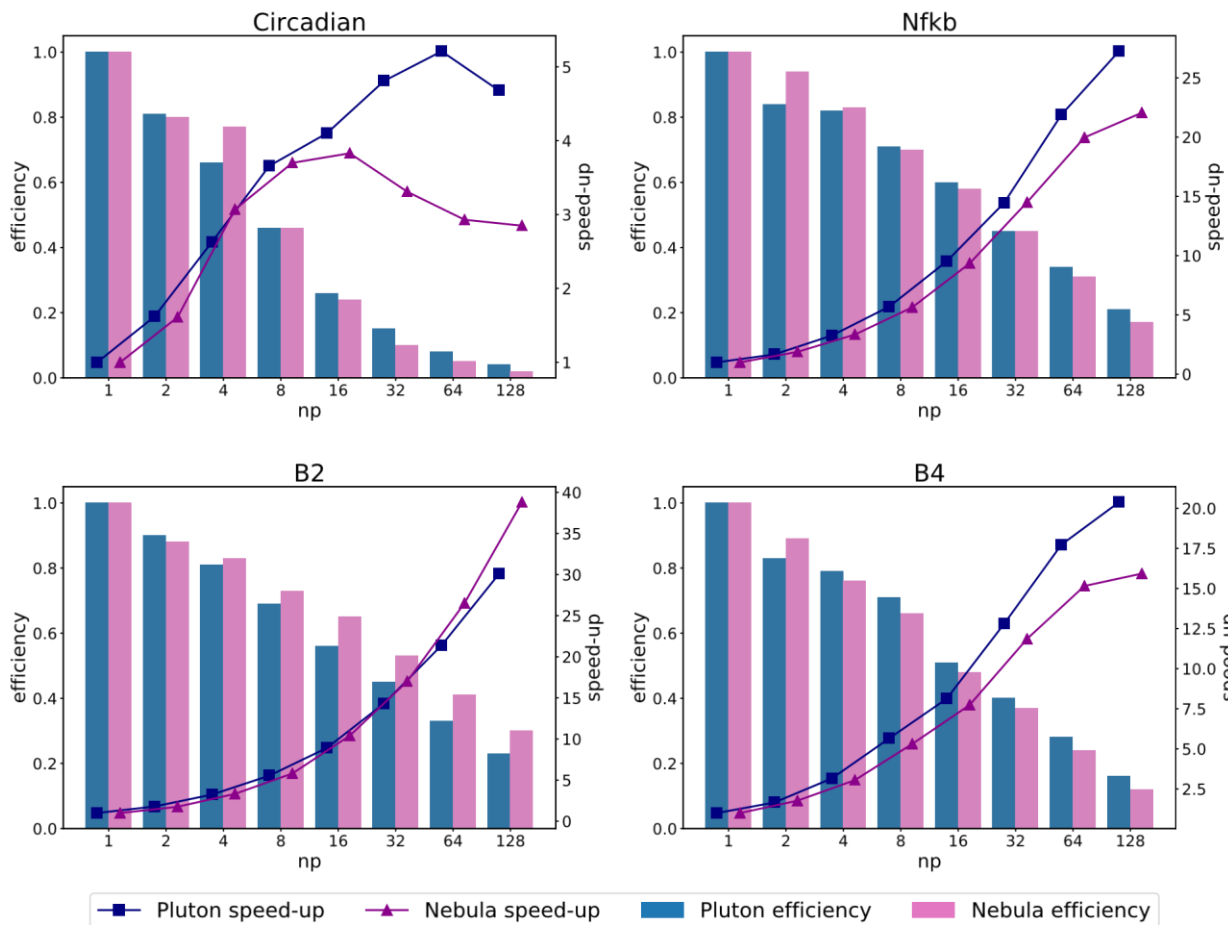


Figure 4: Efficiency and speed-up (calculated as T_{seq}/T_{par}) for every benchmark.

overhead. Furthermore, on *Nebula*, the speedup achieved by B2 is larger than in *Pluton*, because B2 has a better *computation time/communication overhead* ratio on that platform. The execution time is larger on *Nebula*, while the communication overhead remains low on both platforms due to the lower number of iterations performed. Additionally, for the opposite reasons to those explained for the Circadian, B2 is the benchmark for which the drop in the number of evaluations per second and core is the slowest (Figure 5).

With regard to Nfkb and B4, they show results that are between those of Circadian and B2, but for different reasons. Nfkb performs few time-consuming fitness evaluations per iteration, while B4 performs as many evaluations per iteration as B2 but needs much less time for each than Nfkb. This also explains that, in terms of evaluations per second and core, Nfkb behaves more like B2, and B4 more like Circadian (Figure 5).

To better illustrate the influence of the *computation time/communication overhead* ratio on the scalability of the benchmarks, the execution time and overhead of the tasks distributed to the cluster nodes for an execution of the map transformation in the SC method (Figure 3) on *Nebula*, are shown in Figures 6 and 7 for benchmarks B2 and B4,

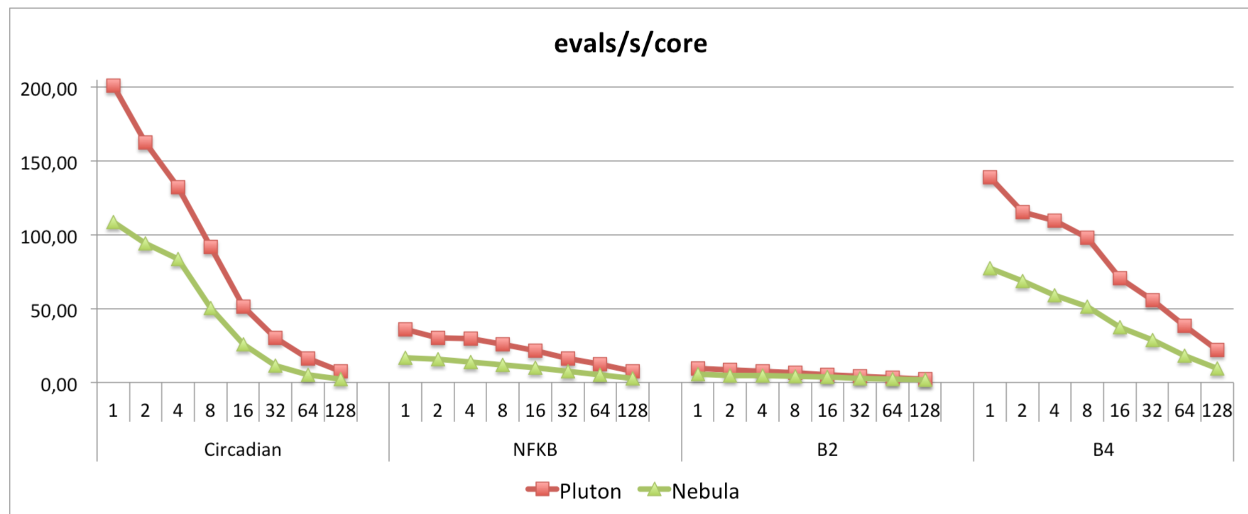
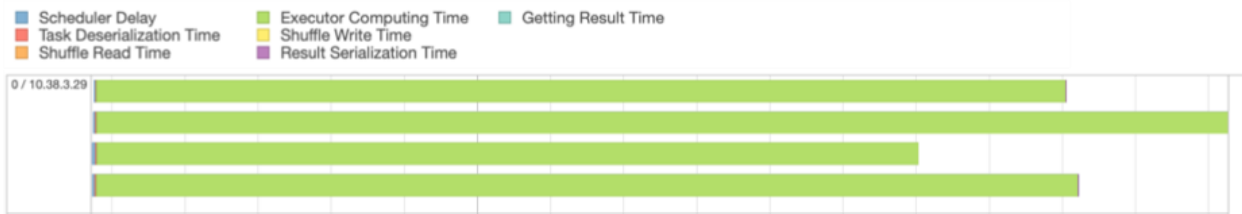


Figure 5: Evaluations per second and core, for every benchmark.

respectively. For each benchmark, two different configurations are shown, a 1-node cluster with 4 cores and an 8-node cluster with 16 cores each (128 cores in total). For each configuration, two types of information are provided: 1) for a node in the cluster, a timing diagram showing the different times involved in the execution of each task; and 2) a table summarizing these times for all tasks in the cluster. In the timing diagram, a bar is used for each task, with green representing the execution time and other colours used for different types of overhead, including task scheduling, task deserialization, and result serialization. Note that the time scale has not been included because it is different in each diagram and is not relevant to the discussion of the results. In the summary table, the time information of all the tasks in the cluster is summarized by providing the minimum, median and maximum values, and two different percentiles. Note that in the table the *Duration* metric measures the computation time and there is an additional metric to measure the garbage collection (GC) overhead.

From Figures 6 and 7 it can be calculated how the *computation time/communication overhead* ratio worsens as the number of cores (tasks) increases. Taking the median values as a reference, the ratio for 4 cores is more than 12 times better than the ratio for 128 cores in the case of B2, and more than 10 times better in the case of B4. Note that, in this particular example, although the median overhead for 128 cores is lower than the median overhead for 4 cores in both benchmarks, the ratio is worse, because the reduction in computation time has also been greater than the reduction in overhead time in both benchmarks. Note also that for B4, the median computing time for 128 cores has fallen below the overhead time (the ratio is less than 1). Furthermore, comparing the ratios for the same number of cores in both benchmarks, the ratio of B2 is more than 21 times better than the ratio of B4 for 4 cores and more than 17 times better for 128 cores.

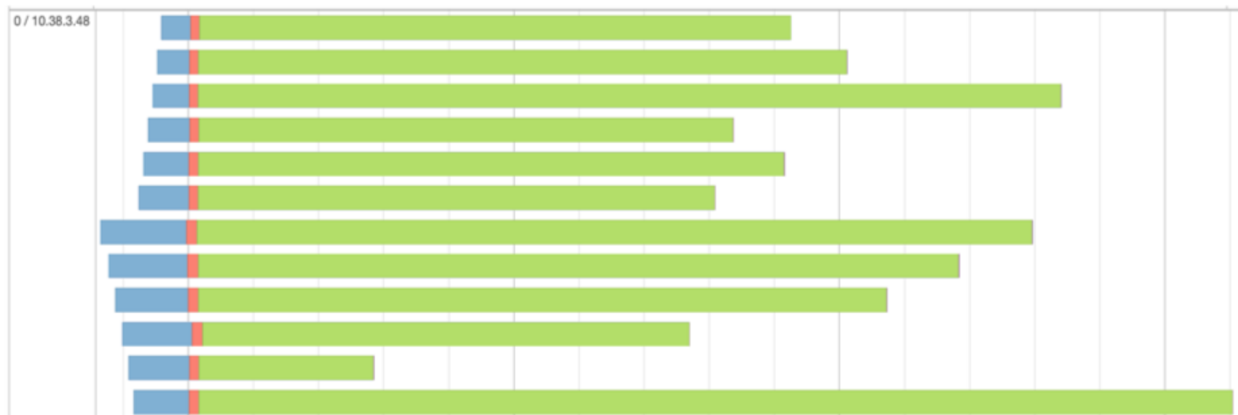
In light of the results presented until now, we can conclude that the problems that benefit most from the parallel computation of fitness evaluations are those that carry out a large number of time-consuming fitness evaluations per



Summary Metrics for 4 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	56 s	1,1 min	1,1 min	1,3 min	1,3 min
Scheduler Delay	0,2 s	0,2 s	0,2 s	0,2 s	0,2 s
Task Deserialization Time	46 ms	80 ms	85 ms	94 ms	94 ms
GC Time	0 ms	41 ms	63 ms	63 ms	63 ms
Result Serialization Time	14 ms	15 ms	16 ms	61 ms	61 ms

a) B2 - 4 cores



Summary Metrics for 128 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	0,3 s	1 s	2 s	3 s	6 s
Scheduler Delay	32 ms	82 ms	0,1 s	0,2 s	0,3 s
Task Deserialization Time	7 ms	21 ms	28 ms	44 ms	70 ms
GC Time	0 ms	0 ms	6 ms	8 ms	21 ms
Result Serialization Time	0 ms	1 ms	1 ms	1 ms	8 ms

b) B2 - 128 cores

Figure 6: Task execution time and overhead for an execution of the map transformation in the SC method (Figure 3) for the B2 problem on *Nebula*. Detail of a cluster node and summary metrics for all tasks in: a) a 1-node cluster with 4 cores; b) an 8-node cluster with 16 cores each (128 cores in total, only 13 cores of one node shown).

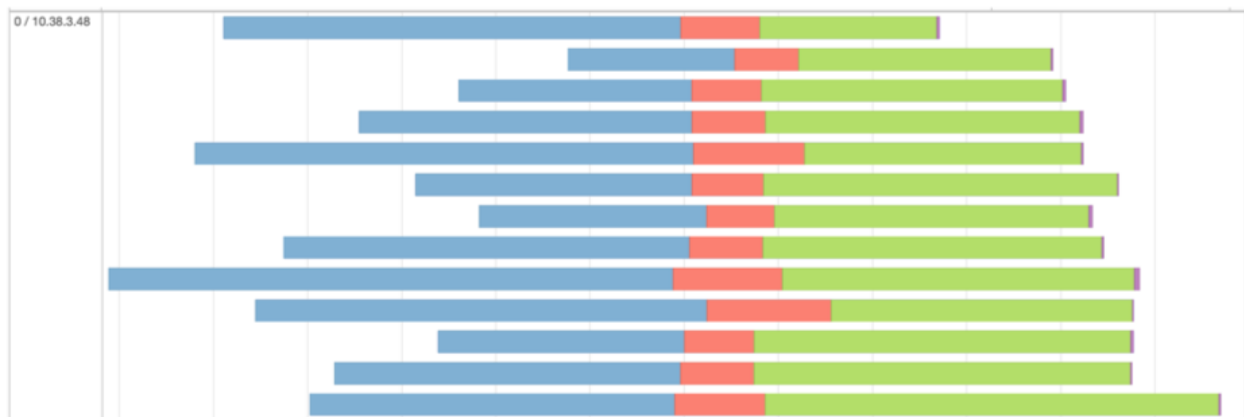
iteration. Moreover, the speed-up and scalability of those problems are further improved on the cloud platform due to a better *computation/communication* ratio. By the contrary, problems that perform a large number of short fitness evaluations per iteration have their scalability limited by the communication overhead.

In this kind of stochastic problems it is also important to evaluate the dispersion of the experimental results. A boxplot for each experiment is shown in Figure 8, where we can see how the parallel method reduces the variability

Summary Metrics for 4 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	4 s	4 s	4 s	5 s	5 s
Scheduler Delay	0,2 s	0,3 s	0,4 s	0,5 s	0,5 s
Task Deserialization Time	33 ms	45 ms	47 ms	52 ms	52 ms
GC Time	0 ms	0 ms	0 ms	0 ms	0 ms
Result Serialization Time	26 ms	28 ms	29 ms	29 ms	29 ms

a) B4 - 4 cores



Summary Metrics for 128 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	74 ms	0,1 s	0,2 s	0,2 s	0,5 s
Scheduler Delay	29 ms	0,1 s	0,2 s	0,2 s	0,3 s
Task Deserialization Time	5 ms	29 ms	39 ms	52 ms	77 ms
GC Time	0 ms	0 ms	0 ms	0 ms	8 ms
Result Serialization Time	0 ms	1 ms	1 ms	2 ms	3 ms

b) B4 - 128 cores

Figure 7: Task execution time and overhead for an execution of the map transformation in the SC method (Figure 3) for the B4 problem on *Nebula*. Detail of a cluster node and summary metrics for all tasks in: a) a 1-node cluster with 4 cores; b) an 8-node cluster with 16 cores each (128 cores in total, only 13 cores of one node shown).

of the execution time with the number of cores. This is an important feature, since it can be used to more accurately predict the boundaries in the cost of resources when using a public cloud.

Figure 9 illustrates with a set of *violinplots* the dispersion in the solution achieved using different number of cores on both platforms *Pluton* and *Nebula* and the median value of *fbest* reported in Table 4. As it was explained in
Section 4, the parallel strategy proposed in this work follows a *master-slave* approach, and therefore no significant

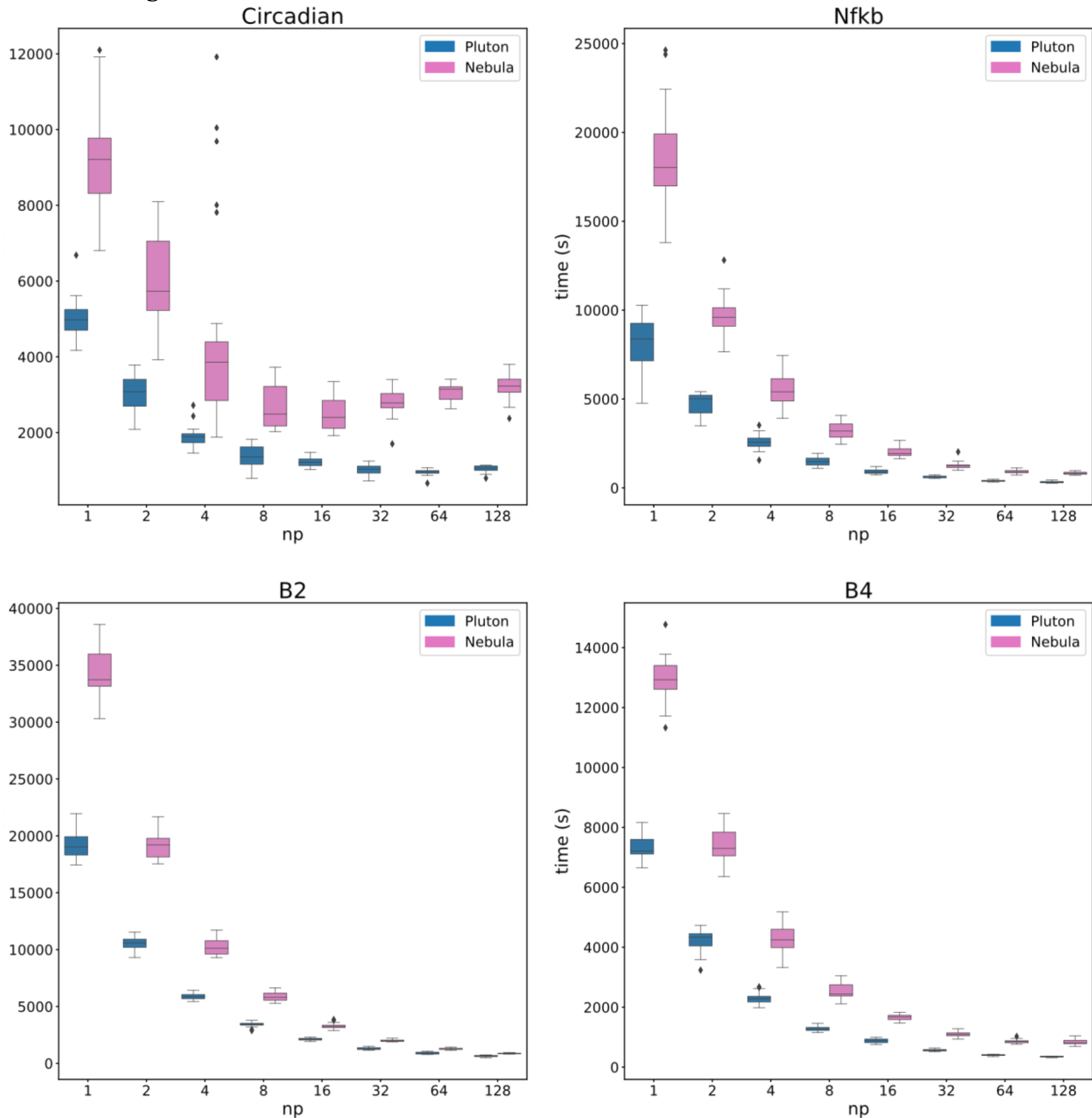


Figure 8: Boxplots that compare the execution times on both platforms, *Nebula* and *Pluton*, for each benchmark.

differences in terms of convergence should be expected for experiments running with a fixed maximum number of evaluations, since this model of parallelism does not modify the systemic properties of the implemented algorithm.

In order to further demonstrate that the convergence remains the same when this parallel strategy is implemented, a

Kruskal-Wallis test [38] was used for each benchmark. We have applied this test to the median *fbest* values reported in

Table 4. In every case, the test was applied three times: once comparing the results obtained in each different platform

(*Pluton* and *Nebula*) and then one final time for the two platforms altogether, since neither the parallelism nor the

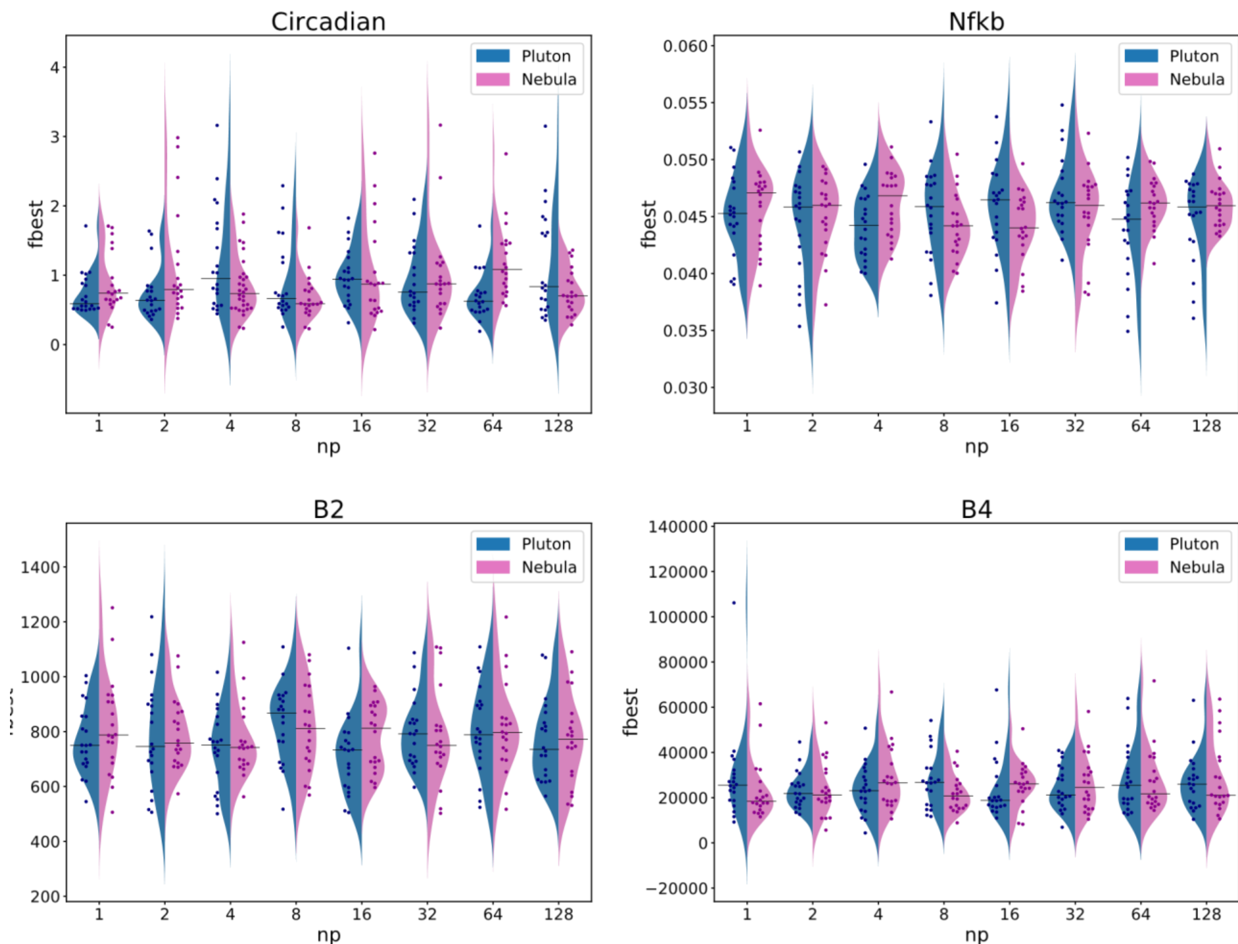


Figure 9: Violinplots showing dispersion of the best value achieved for each benchmark on both platforms. A horizontal bar representing median *fbest* is also included.

Table 5: Kruskal-Wallis H test with $k-1 = 7$ degrees of freedom and a significance level $\alpha = 0.05$. The third column *Total* shows the results of the

test considering all runs in both platforms.

	Pluton		Nebula		Total	
	H	*p*-value	*H*	*p*-value	*H*	*p*-value
Circadian	14.99935	0.03601	16.37765	0.02188	8.10691	0.32326
Nfkb	7.43054	0.38547	11.52554	0.11728	4.16831	0.76020
B2	9.76691	0.20217	2.64885	0.91548	9.13200	0.24332
B4	3.57033	0.82772	6.41152	0.49260	2.43737	0.93174

infrastructure should modify the behaviour of the sequential algorithm in terms of convergence for this implementation model. The results are shown in Table 5. We can see that for every experiment performed, the *p-value* is over the significance level except for the case of Circadian benchmark running on *Pluton* and *Nebula* separately. This could be due to the fact that, as it can be seen in Figure 9, this benchmark presents a large dispersion in the experimental test, and thus more runs of each experiment should be performed to get more representative results from the statistical analysis. In fact, when the medians of all the 40 executions on both infrastructures are considered, a *p-value* over the significance level for this benchmark is also obtained.

To conclude this section, we would like to make some comments on the benefits of using the Spark framework, based on our own experience with the proposed parallelization of the eSS metaheuristic. Readers who are more interested in the performance evaluation of the Spark framework can refer to [70, 64, 42, 54]. Although Spark's runtime performance is worse than that of other classic HPC approaches, such as MPI or OpenMP, it provides some features that may be especially appealing to those who, like us, are interested in parallelizing and scaling their algorithms in the Cloud. These features include high-level programming abstractions and intrinsic support for data distribution, fault tolerance and load balancing.

Most of the advantages that Spark provides in the implementation of iterative distributed algorithms such as eSS, come from using the RDD abstraction to represent the algorithm data. Formally, an RDD is a read–only fault–tolerant partitioned collection of records. RDDs are created by using the *parallelize* method, which automatically partitions and distributes data among available worker nodes, or by applying transformations to other RDDs. Spark provides a rich programming interface for manipulating RDDs using transformations such as *map*, *filter* or *join*, which perform the same operation on many distributed data elements at the same time, and actions such as *count*, *reduce* or *collect*, which return a value from an RDD to the application. There are also methods to control the partitioning of RDDs, such as *partitionBy* or *repartition*, and to explicitly persist RDDs in memory and on disk, such as *cache* and *persist*.
Compared to other parallel programming models, such as message passing, RDDs are significantly easier to program.

Moreover, RDDs include fault tolerance as standard. The transformations to compute an RDD are pipelined to form a *lineage* that is computed lazily the first time the RDD is used in an action.

If any RDD partition is lost, it can be recomputed from its lineage at any time. This guarantees that executions are completed successfully even in the event of failures, which is specially useful for long-running applications. For example, in the experiments in this paper, a small number of runs were affected by faulty worker nodes. Even so, all executions concluded successfully, albeit with an execution time penalty. Because the Spark framework handles all the details in a transparent manner, developers are freed from the burden of implementing fault tolerance in their programs.

Finally, data distribution and load balancing are other features that are also automatically handled by the Spark framework. When programming with Spark, developers write a *driver* program in which they create, partition and distribute RDDs, apply transformations and invoke actions on them. Whenever an action is executed on an RDD, the Spark scheduler uses the RDD lineage to launch tasks that compute the missing RDD partitions. These tasks are assigned to the worker nodes taking into account data locality and, at each worker node, the tasks are scheduled among all available cores. In this way, Spark balances the internal load of the nodes. In addition, the developer can improve load balancing by setting an appropriate number of RDD partitions. It is recommended to use a number of partitions between 2 and 4 times the number of available cores, although the actual value depends on the application. In this paper, a number of partitions equal to the number of available cores in the cluster has been used for all the experiments, in order to avoid biases in the results.

6. Concluding Remarks

This article describes in detail a parallelization of the enhanced Scatter Search metaheuristic using Spark. The parallel program is a particularization of a general software framework developed to support different Scatter Search implementations. This parallelization follows a master-slave approach, where a single monolithic population is used and the fitness function evaluations are distributed among Spark workers.

The proposal was thoroughly evaluated on two different platforms, a cluster and an *OpenNebula* cloud platform, using a representative set of parameter estimation problems in Computational Systems Biology. The experimental results show that parallel implementation achieves good results by accelerating the solution of all the benchmark problems without modifying the systemic properties of the original algorithm. The problems that benefit most from parallelization are those that perform a large number of time-consuming fitness evaluations per iteration. On the contrary, for those problems that perform a small number of short fitness evaluations per iteration, the results show that their scalability is limited. Another important result of the parallel implementation versus the sequential one is the reduction in the dispersion of results when the number of slaves increases.

Although the parallel implementation of the enhanced Scatter Search was developed and tested for parameter estimation problems in computational systems biology, it could also be applied to solve global optimization problems in other fields. Moreover, the experimental results and findings presented in this paper could be useful for those interested in the potential of the Spark framework to implement parallel metaheuristics and may guide the proposal of new parallel strategies for the Scatter Search or other population-based metaheuristics.

As future work, we will explore other parallelization approaches, i.e. an island-based model, that modify the behavior of the original sequential algorithm.

References

[1] Agarwal, D., Karamati, S., Puri, S., and Prasad, S. K. Towards an mpi-like framework for the azure cloud platform. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on* (2014), IEEE, pp. 176–185.

[2] Alba, E., and Luque, G. Evaluation of parallel metaheuristics. In *PPSN-EMAA'06* (Reykjavik, Iceland, September 2006), pp. 9–14.

[3] Alba, E., Luque, G., and Nesmachnow, S. Parallel metaheuristics: recent advances and new trends. *International Transactions in Operational Research 20*, 1 (2013), 1–48.

[4] Amazon Web Services. https://aws.amazon.com.

[5] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., and Stoica, I. A view of cloud computing. *Communications of the ACM 53*, 4 (2010), 50–58.

[6] Ashish, T., Kapil, S., and Manju, B. Parallel bat algorithm-based clustering using mapreduce. In *Networking Communication and Data Knowledge Engineering* (2018), G. M. Perez, K. K. Mishra, S. Tiwari, and M. C. Trivedi, Eds., pp. 73–82.

[7] Balsa-Canto, E., Banga, J. R., Egea, J. A., Fernandez-Villaverde, A., and de Hijas-Liste, G. M. Global optimization in systems biology: Stochastic methods and their applications. In *Advances in Systems Biology* (2012), I. I. Goryanin and A. B. Goryachev, Eds., pp. 409–424.

[8] Banga, J. R. Optimization in computational systems biology. *BMC Systems Biology 2*, 1 (2008), 47.

[9] Bao, R., Huang, L., Andrade, J., Tan, W., Kibbe, W. A., Jiang, H., and Feng, G. Review of current methods, applications, and data management for the bioinformatics analysis of whole exome sequencing. *Cancer Informatics 13s2* (2014), CIN.S13779.

[10] Bevilacqua-Linn, M. *Functional Programming Patterns in Scala and Clojure: Write Lean Programs for the JVM*. Pragmatic programmers. Pragmatic Bookshelf, 2013.

[11] Boukouvala, F., Misener, R., and Floudas, C. A. Global optimization advances in mixed-integer nonlinear programming, minlp, and constrained derivative-free optimization, cdfo. *European Journal of Operational Research 252*, 3 (2016), 701–727.

[12] Bozejko˙ , W., and Wodecki, M. Parallel path-relinking method for the flow shop scheduling problem. In *International Conference on Computational Science* (2008), Springer, pp. 264–273.

[13] Buyya, R., Broberg, J., and Goscinski, A. M. *Cloud computing: Principles and paradigms*, vol. 87. John Wiley & Sons, 2010.

[14] Centro de Supercomputacion de´ Galicia (CESGA). http://www.cesga.es.

[15] Chun-Wei Tsai, Heng-Ci Chang, Kai-Cheng Hu, and Ming-Chao Chiang. Parallel coral reef algorithm for solving jsp on spark. In *2016 IEEE International Conference on Systems, Man, and Cybernetics (SMC)* (2016), pp. 001872–001877.

[16] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *Communication ACM 51*, 1 (2008), 107–113.

[17] Dennis Jr, J. E., Gay, D. M., and Welsch, R. E. Algorithm 573: Nl2sol an nonlinear least-squares algorithm. *ACM Transactions on Mathematical Software (TOMS) 7*, 3 (1981), 369–383.

[18] Egea, J. A., Balsa-Canto, E., Garc´ıa, M. S. G., and Banga, J. R. Dynamic optimization of nonlinear processes with an enhanced scatter search method. *Industrial & Engineering Chemistry Research 48*, 9 (2009), 4388–4401.

[19] Egea, J. A., Mart´ı, R., and Banga, J. R. An evolutionary method for complex-process optimization. *Computers & Operations Research 37*, 2 (2010), 315–324.

[20] Egea, J. A., Rodr´ıguez-Fernandez´ , M., Banga, J. R., and Mart´ı, R. Scatter search for chemical and bio-process optimization. *Journal of Global Optimization 37*, 3 (2007), 481–503.

[21] Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (2010), ACM, pp. 810–818.

[22] Evangelinos, C., and Hill, C. Cloud computing for parallel scientific HPC applications: Feasibility of running coupled atmosphere-ocean climate models on amazon's EC2. In *1st Workshop on Cloud Computing and its Applications (CCA'08)* (2008), pp. 1–6.

[23] Exposito´ , R. R., Taboada, G. L., Ramos, S., Tourino˜ , J., and Doallo, R. Performance analysis of HPC applications in the cloud. *Future Generation Computer Systems 29*, 1 (2013), 218–229.

[24] Floudas, C. A., and Gounaris, C. E. A review of recent advances in global optimization. *Journal of Global Optimization 45*, 1 (2009), 3–38. [25] Gabor´ , A., and Banga, J. R. Robust and efficient parameter estimation in dynamic models of biological systems. *BMC Systems Biology 9*, 1 (2015), 74.

[26] Garcıa-Lopez´ , F., Melian´ -Batista, B., Moreno-Perez´ , J. A., and Moreno-Vega, J. M. Parallelization of the scatter search for the p-median problem. *Parallel Computing 29*, 5 (2003), 575–589.

[27] Gittens, A., Devarakonda, A., Racah, E., Ringenburg, M., Gerhardt, L., Kottalam, J., Liu, J., Maschhoff, K., Canon, S., Chhugani, J., Sharma, P., Yang, J., Demmel, J., Harrell, J., Krishnamurthy, V., Mahoney, M. W., and Prabhat. Matrix factorizations at scale: A comparison of scientific data analytics in spark and c+mpi using three case studies. In *2016 IEEE International Conference on Big Data (Big Data)* (2016), pp. 204–213.

[28] Glover, F. Heuristics for integer programming using surrogate constraints. *Decision Sciences 8*, 1 (1977), 156–166.

[29] Glover, F. A template for scatter search and path relinking. In *Selected Papers from the Third European Conference on Artificial Evolution* (1998), Springer, pp. 3–54.

[30] Glover, F., Laguna, M., and Mart´ı, R. Fundamentals of scatter search and path relinking. *Control and Cybernetics 39* (2000), 653–684.

[31] Hansen, N., Auger, A., Finck, S., and Ros, R. Real-parameter black-box optimization benchmarking 2010: Experimental setup. Tech. Rep. Rapports de Recherche RR-6828, Institut National de Recherche en Informatique et en Automatique (INRIA), 2009.

[32] Hazelhurst, S. Scientific computing using virtual high-performance computing: a case study using the amazon elastic computing cloud. In *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology* (2008), ACM, pp. 94–103.

[33] Huang, D., and Lin, J. Scaling populations of a genetic algorithm for job shop scheduling problems using mapreduce. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science* (2010), pp. 780–785.

[34] Jackson, K. R., Ramakrishnan, L., Muriki, K., Canon, S., Cholia, S., Shalf, J., Wasserman, H. J., and Wright, N. J. Performance analysis of high performance computing applications on the amazon web services cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on* (2010), IEEE, pp. 159–168.

[35] Jin, C., Vecchiola, C., and Buyya, R. MRPGA: an extension of MapReduce for parallelizing genetic algorithms. In *IEEE Fourth International Conference on eScience, eScience'08* (2008), IEEE, pp. 214–221.

[36] Kamburugamuve, S., Wickramasinghe, P., Ekanayake, S., and Fox, G. C. Anatomy of machine learning algorithm implementations in mpi, spark, and flink. *The International Journal of High Performance Computing Applications 32*, 1 (2018), 61–73.

[37] Keahey, K., Freeman, T., Lauret, J., and Olson, D. Virtual workspaces for scientific applications. *Journal of Physics: Conference Series 78*, 1 (2007), 012038.

[38] Kruskal, W. H., and Wallis, W. A. Use of ranks in one-criterion variance analysis. *Journal of the American Statistical Association 47*, 260

(1952), 583–621.

[39] Laguna, M., Mart´ı, R., and Mart´ı, R. *Scatter Search: Methodology and Implementations in C*. Operations Research/Computer Science

Interfaces Series. Springer US, 2003.

[40] Lee, W., Hsiao, Y., and Hwang, W. Designing a parallel evolutionary algorithm for inferring gene networks on the cloud computing environment. *BMC Systems Biology 8*, 1 (2014), 5.

[41] Li, J., Humphrey, M., Van Ingen, C., Agarwal, D., Jackson, K., and Ryu, Y. escience in the cloud: A modis satellite data reprojection and reduction pipeline in the windows azure platform. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on* (2010), IEEE, pp. 1–10.

[42] Li, M., Tan, J., Wang, Y., Zhang, L., and Salapura, V. Sparkbench: A spark benchmarking suite characterizing large-scale in-memory data

analytics. *Cluster Computing 20*, 3 (Sept. 2017), 2575–2589.

[43] Lipniacki, T., Paszek, P., Brasier, A., Luxon, B., and Kimmel, M. Mathematical model of nf-κb regulatory module. *Journal of Theoretical

Biology 228*, 2 (2004), 195–215.

[44] Locke, J., Millar, A., and Turner, M. Modelling genetic networks with noisy and varied experimental data: The circadian clock in arabidopsis thaliana. *Journal of Theoretical Biology 234*, 3 (2005), 383–393.

[45] Lopez´ , F. G., Torres, M. G., Batista, B. M., Perez´ , J. A. M., and Moreno-Vega, J. M. Solving feature subset selection problem by a parallel scatter search. *European Journal of Operational Research 169*, 2 (2006), 477–489.

[46] Lu, X., Liang, F., Wang, B., Zha, L., and Xu, Z. DataMPI: extending MPI to Hadoop-like big data computing. In *Parallel and Distributed

Processing Symposium, 2014 IEEE 28th International* (2014), IEEE, pp. 829–838.

[47] Lu, Y., Cao, B., Rego, C., and Glover, F. A tabu search based clustering algorithm and its parallel implementation on spark. *Applied Soft

Computing 63* (2018), 97 – 109.

[48] McNabb, A. W., Monson, C. K., and Seppi, K. D. Parallel PSO using MapReduce. In *IEEE Congress on Evolutionary Computation, CEC2007*

(2007), IEEE, pp. 7–14.

[49] Microsoft Azure. https://azure.microsoft.com.

[50] Napper, J., and Bientinesi, P. Can cloud computing reach the top500? In *Proceedings of the combined workshops on UnConventional high

performance computing workshop plus memory access workshop* (2009), ACM, pp. 17–20.

[51] Odersky, M., Spoon, L., and Venners, B. *Programming in Scala*. Artima, 2008.

[52] OpenNebula by OpenNebula Systems. http://www.opennebula.org.

[53] Ostermann, S., Iosup, A., Yigitbasi, N., Prodan, R., Fahringer, T., and Epema, D. An early performance analysis of cloud computing services

for scientific computing. Tech. Rep. PDS-2008-006, Delft University of Technology, 2008.

[54] Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., and Chun, B.-G. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (USA, 2015), NSDI'15, USENIX Association, p. 293–307.

[55] Penas, D. R., Gonzalez´ , P., Egea, J. A., Banga, J. R., and Doallo, R. Parallel metaheuristics in computational biology: An asynchronous cooperative enhanced scatter search method. *Procedia Computer Science 51*, 0 (2015), 630 – 639. International Conference On Computational Science (ICCS 2015).

[56] Penas, D. R., Gonzalez´ , P., Egea, J. A., Doallo, R., and Banga, J. R. Parameter estimation in large-scale systems biology models: a parallel

and self-adaptive cooperative strategy. *BMC Bioinformatics 18*, 1 (2017), 52.

[57] Radenski, A. Distributed simulated annealing with MapReduce. In *Proceedings of the 2012T European Conference on Applications of*

Evolutionary Computation. Springer, 2012, pp. 466–476.

[58] Ramakrishnan, L., Jackson, K. R., Canon, S., Cholia, S., and Shalf, J. Defining future platform requirements for e-science clouds. In

Proceedings of the 1st ACM symposium on Cloud computing (2010), ACM, pp. 101–106.

[59] Reali, F., Priami, C., and Marchetti, L. Optimization algorithms for computational systems biology. *Frontiers in Applied Mathematics and*

Statistics 3 (2017), 6.

[60] Reed, D. A., and Dongarra, J. Exascale computing and big data. *Communications of the ACM 58*, 7 (2015), 56–68.

[61] Resende, M. G., Ribeiro, C. C., Glover, F., and Mart´ı, R. *Scatter Search and Path-Relinking: Fundamentals, Advances, and Applications*. Springer US, Boston, MA, 2010, pp. 87–107.

[62] Reyes-Ortiz, J. L., Oneto, L., and Anguita, D. Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf. *Procedia*

Computer Science 53 (2015), 121 – 130.

[63] Rodriguez-Fernandez, M., Egea, J. A., and Banga, J. R. Novel metaheuristic for parameter estimation in nonlinear dynamic biological

systems. *BMC Bioinformatics 7*, 1 (2006), 483.

[64] Shi, J., Qiu, Y., Minhas, U. F., Jiao, L., Wang, C., Reinwald, B., and O¨ zcan, F. Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proc. VLDB Endow. 8*, 13 (Sept. 2015), 2110–2121.

[65] Teijeiro, D., Pardo, X. C., Gonzalez´ , P., Banga, J. R., and Doallo, R. Implementing parallel differential evolution on spark. In *Applications of Evolutionary Computation - 19th*

European Conference, EvoApplications 2016, Porto, Portugal, March 30 - April 1, 2016, Proceedings, Part II (2016), G. Squillero and P. Burelli, Eds., vol. 9598 of *Lecture Notes in Computer Science*, Springer, pp. 75–90.

[66] Teijeiro, D., Pardo, X. C., Gonzalez´ , P., Banga, J. R., and Doallo, R. Towards cloud-based parallel metaheuristics. *Int. J. High Perform.*

Comput. Appl. 32, 5 (2018), 693–705.

[67] Teijeiro, D., Pardo, X. C., Penas, D. R., Gonzalez´ , P., Banga, J. R., and Doallo, R. Evaluation of parallel differential evolution implementations on mapreduce and spark. In *Euro-Par 2016: Parallel Processing Workshops - Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers* (2016), F. Desprez, P. Dutot, C. Kaklamanis, L. Marchal, K. Molitorisz, L. Ricci, V. Scarano, M. A. Vega-Rodr´ıguez, A. L. Varbanescu, S. Hunold, S. L. Scott, S. Lankes, and J. Weidendorfer, Eds., vol. 10104 of *Lecture Notes in Computer Science*, Springer, pp. 397–408.

[68] Tsai, C.-W., Liu, S.-J., and Wang, Y.-C. A parallel metaheuristic data clustering framework for cloud. *Journal of Parallel and Distributed*

Computing 116 (2018), 39 – 49.

[69] Zˇilinskas, A., and Zhigljavsky, A. Stochastic global optimization: A review on the occasion of 25 years of informatica. *Informatica 27*, 2

(2016), 229–256.

[70] Veiga, J., Exposito´ , R. R., Pardo, X. C., Taboada, G. L., and Tourino˜ , J. Performance evaluation of big data frameworks for large-scale data analytics. In *2016 IEEE International Conference on Big Data (Big Data)* (2016), pp. 424–431.

[71] Verma, A., Llora, X., Goldberg, D. E., and Campbell, R. H. Scaling genetic algorithms using MapReduce. In *2009 Ninth International*

Conference on Intelligent Systems Design and Applications (2009), IEEE, pp. 13–18.

[72] Villaverde, A. F., Egea, J. A., and Banga, J. R. A cooperative strategy for parameter estimation in large scale systems biology models. *BMC*

Systems Biology 6, 1 (2012), 75.

[73] Villaverde, A. F., Frohlich¨ , F., Weindl, D., Hasenauer, J., and Banga, J. R. Benchmarking optimization methods for parameter estimation

in large kinetic models. *Bioinformatics 35*, 5 (2019), 830–838.

[74] Villaverde, A. F., Henriques, D., Smallbone, K., Bongard, S., Schmid, J., Cicin-Sain, D., Crombach, A., Saez-Rodriguez, J., Mauch, K., Balsa-Canto, E., Mendes, P., Jaeger, J., and Banga, J. R. Biopredyn-bench: a suite of benchmark problems for dynamic modelling in systems
biology. *BMC Systems Biology 9*, 1 (2015), 1–15.

[75] Wu, B., Wu, G., and Yang, M. A mapreduce based ant colony optimization approach to combinatorial optimization problems. In *2012 8th*

International Conference on Natural Computation (2012), pp. 728–732.

[76] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (2012), USENIX Association, pp. 2–2.
[77] Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., Ghodsi, A., Gonzalez, J., Shenker, S., and Stoica, I. Apache spark: A unified engine for big data processing. *Commun. ACM 59*, 11 (Oct. 2016), 56–65.
[78] Zhou, C. Fast parallelization of differential evolution algorithm using MapReduce. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation* (2010), ACM, pp. 1113–1114.